Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e Estatística
INE5416 – Paradigmas de Programação
Graduando 1: Cainã Correa Caldas
Graduando 2: Vinicius Guedes dos Santos

Trabalho I - Haskell

O jogo escolhido pela nossa dupla foi o Wolkenkratzer, o enunciado pedia que se utiliza-se a linguagem Haskell para criar um programa capaz de encontar a solução de qualquer cenário possível do jogo, limitado aos cenários de tamanho menor ou igual a 6x6.

A solução que encontramos foi utilizar um *backtracking* simples, onde as posições do tabuleiro são preenchidas da esquerda para a direita e de cima para baixo com números em ordem crescente, quando um número é colocado em uma posição, testa-se se alguma regra é quebrada, caso isso aconteça, o programa tenta outro número, ou volta e altera a posição anterior. Caso nenhuma regra seja quebrada, o número testado é mantido na posição e o algoritmo avança para a próxima posição. Para representar o tabuleiro, utilizamos uma lista que através de algumas funções pode ser indexada como uma matriz, ou seja, para acessar a posição *(i, j)* da matriz, nosso programa acessa a posição *i + tam(m).j* da lista, onde *tam(m)* é uma função que retorna o número de linhas da matriz. Além das posições vazias, a referida matriz também contém, as informações a respeito de quantos prédios podem ser vistos de uma certa posição. Durante a execução do *backtracking,* o programa utilizará três diferentes estruturas de dados, uma delas é o tabuleiro propriamente dito, essa matriz será modificada durante o processo e seu estado final é a solução do problema, a outra estrutura utilizada é uma matriz que mantém informações a respeito de que números já foram testados em cada posição, e por fim, uma pequena lista que contém todos os números possíveis de um determinado cenário.

Alguns trechos de código importantes:

```
resolve :: Int-> Int -> Int -> [Int] -> [Int] -> [Int] -> Bool -> [Int]
resolve k x y m v p d -- k= limiteDaRecursao, x, y, m, v=matrizGuardaIndexNoVetorDePossiveis, p=listaDeNumerosPossiveis[100% constante]
   | k <=0 = m -- ESCOLHA AQUI v OU m
   | y < 0 = m -- foi sinalizado y = -1 -> encerrar execução .. na vdd nao tah parando mas era a ideia
   --tudo certo, vamo pro proximo
   -- Estava indo e encontrou posicao que na pode mudar, vai para a proxima
   | (getxym x y v == -1 && d == indo) = resolve (k-1) (nextX x m) (nextY x y m) m v p indo
   -- Estava voltando e encontrou posicao que nao pode mudar, volta mais uma posicao
   | (getxym x y v == -1 && d == voltando) = resolve (k-1) (backX x m) (backY x y) m v p voltando
   -- nenhum encaixa aki, mude o anterior (mp pega o ultimo elemento -> o maior possível)
   | (getxym x y m) >= (mp p) = resolve (k-1) (backX x m) (backY x y) (setXY o x y m) (setXY 0 x y v) p voltando
   --tudo certo, substitui e bola pra frente
    | tahOk x y (setXY (p!!(getxym x y v)) x y m ) = resolve (k-1) (nextX x m) (nextY x y m) (setXY (p!!(getxym x y v)) x y m ) (setXY ((getxym x y v) +1) x y v )
p indo
   -- tenta o proximo numero aki
   | not (tahOk x y (setXY (p!!(getxym x y v)) x y m )) = resolve (k-1) x y (setXY (p!!(getxym x y v)) x y m ) (setXY (( (getxym x y v) +1) ) x y v ) p indo
```

A funçao *resolve* implementa o *backtracking*.

```
tahOk :: Int -> Int -> [Int] -> Bool
tahOk x y m
     | (getxym 0 0 m == d) && ((x == y) || (x + y == (tam m) - 1)) = (vejoCerto x y m) && (not (jaTemNaLinha (getxym x y m) x y (setXY (o) x y m) ) ) && (not
(jaTemNaColuna (getxym x y m) x y (setXY (o) x y m)))
        && (not (jaTemNasDiagonais (getxym x y m) x y (setXY (o) x y m)))
     | otherwise = (vejoCerto x y m) && (not (jaTemNaLinha (getxym x y m) x y (setXY (o) x y m))) && (not (jaTemNaColuna (getxym x y m) x y (setXY (o) x y
m)) )
```

A função *tahOk* verifica se a matriz no estado atual está quebrando alguma regra, esta informação é utilizada pela função *resolve* para saber se é adequado manter um número naquela posição ou se deve-se tentar outro. Esta função chama outras que verificam a corretude das linhas, colunas e possivelmente diagonais.

Para informar a entrada, é preciso alterar o código fonte, adicionando o problema na forma de lista, esta lista deverá conter os espaços em branco “o” onde o cenário será resolvido e também os números das laterais, que indicam quantos prédios devem ser vistos de uma certa posição, onde não houver número nenhum, deve-se colocar o valor da constante “*e”*. Além disso, para indicar que a solução de um determinado cenário deve levar em consideração as diagonais para avaliar repetiçõos de número, deve-se preencher a primeira posição da lista que representa o cenário com a constante *d*, caso contrário, qualquer valor nesta posição será considerado como um cenário em que não se considera as diagonais. Para indicar quais valores podem ser utilizados em um cenário, deve-se criar uma outra lista contendo tais valores. A solução do cenário será impressa na tela após a execução, mas além disso, o código tem alguns exemplos sobre como comparar a solução encontrada com àquela esperada e imprimir na tela se são iguais, isso facilita o teste da solução de várias matrizes de uma vez só.

Algumas das dificuldades encontradas foram a solução de cenários em que o tabuleiro já inicia com alguma posição preenchida e dos cenários em que não há nenhuma restrição quanto ao número de prédios vistos de uma posição. A solução do primeiro problema foi analisar a matriz principal previamente as posições equivalentes na matriz de tentativas, matriz esta que indica qual número deve-se tentar agora, dessa forma, toda vez que o algoritmo passa por uma posição já preenchida ele a ignora. A outra dificuldade foi resolvida de maneira similar.